In *Constructing Programs from Specifications,* B. Möller (ed.),
Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1991, pp. 225-242

Transformational Implementation of Historical Reference

Martin S. Feather[1]

USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292, USA
feather@isi.edu

Abstract

'Historical reference' is the ability to refer to information in prior states of a program's computation history. We use program transformation to achieve reasonably efficient implementations of specifications making use of historical reference. These implementations remember sufficient information as they execute, and use conventional queries of this remembered information in place of the historical references of the original specifications.

We show how the design of these transformations exemplifies issues common to transformational implementation – choice between special- and general-purpose transformations, layering of transformations and normal forms, transformational definition of language extensions, and targeting transformations to take advantage of the capabilities of the target language.

1. INTRODUCTION

Historical reference means the ability to refer to information in prior states of the computation history. For example, in an imperative program, it would mean reference to past values of data structures. Historical reference is not usually provided as a programming language concept because in general it requires that all prior information be retained from state to state – this would imply unacceptably large storage space costs. Specification languages are under no such limitations, so they often provide historical reference, indeed, they often provide *temporal* reference - the ability to refer to information in not only past states, but also future states. Herein we will consider only historical reference; implementation of future reference appears very hard.

This paper shows how program transformation can be used to achieve reasonably efficient implementations of specifications that make use of historical reference, and discusses some general issues of program transformation that are exemplified by this particular effort. The paper is organized as follows:

section 2 describes what we mean by 'historical reference', and shows how it is provided within our specification language.

[1] The author has been supported in part by Defense Advanced Research Projects Agency grant No. NCC-2-520, and in part by Rome Air Development Center contract No. F30602-85-C-0221 and F3060289-C-0103. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, RADC, NSF, the U.S. Government, or any other person or agency connected with them.

section 3 presents the general-purpose transformation that we have built for automatically removing all instances of historical reference in a specification. The objective of this transformation is to produce a reasonably efficient implementation of historical reference.

section 4 considers some special cases of historical reference that admit to more efficient implementation than that provided by our general-purpose transformation. We show how special purpose transformations can handle these, and then discuss the tradeoffs between the alternatives of using a suite of such special case transformations and using follow-on transformations to optimize the code produced by the general-purpose historical removal transformation.

section 5 shows how we can write transformations that take advantage of a key feature of our target language. We show how transformations can produce code that use this feature, and discuss the benefits and disadvantages of this alternative.

section 6 outlines a way of providing alternative notations for specification of historical reference. These are defined by transformation, i.e., translated into the 'kernel' form of historical reference. We discuss why it is appropriate to leave the historical reference removal transformations operating on the kernel language.

section 7 reviews related work in the use and transformation of historical reference, and in issues of program transformation.

section 8 concludes with a summary of the status of our transformations and the lessons that we have drawn from having built them.

2. HISTORICAL REFERENCE - A 'SPECIFICATION FREEDOM'

The naive implementation of historical reference – remembering *all* information in the computation history – is inappropriate even for prototyping or simulation of the specification on any but the simplest of test cases, let alone for an efficient implementation. In order to achieve efficient implementations it is necessary to convert a specification that uses historical reference into one which remembers sufficient but not unnecessarily much information as it executes, and to convert its historical references into references of this remembered information. Historical reference is thus a 'specification freedom' as termed in [14], namely the freedom of specifications to make arbitrary reference to historical information without regard to how this will be implemented efficiently.

2.1 Our specification language

Our studies have been conducted in the framework an imperative-style state-query specification language, Gist [4]. Our target language is AP5 [8], a database oriented extension of commonlisp [19]. We have a translation mechanism which converts specifications written in a subset of Gist into AP5, where the subset is that which has a direct counterpart in AP5. Historical reference in particular has no counterpart, and so must be removed by transformation prior to this translation. We will describe only the aspects of these languages relevant to the purposes of this paper.

2.1.1 Relational data structures

In Gist, the only persistent data structures are relations among atomic objects - more complex structures can be represented by such relations. We hope that it is clear that our results could be extended to a language with more complex persistent data structures. AP5 too has relations, indeed the bulk of AP5 serves to extend commonlisp with a relational database like capability.

Relations in both Gist and AP5 are declared with the types of the objects that they relate. For example, if specifying a lending-library, we might declare a binary relation (relations can be of arbitrary arity, not just binary) has-book to hold between objects of type person and type book. This relation can be populated by what we call *tuples*, in this case of arity 2 and whose first object is of type person and whose second object is of type book. Relations can be queried - that is, we may ask whether the tuple of a given person and book is in the relation. If in the specification `p` denotes an object of type person, and `b` denotes an object of type book, we would write

```
has-book(p,b)
```

to query whether or not the tuple `p,b` is in the has-book relation.

The tuples in a given relation can be changed by inserting tuples into the relation and by removing tuples that are already in the relation. We write

```
insert has-book(p,b)
```

as a statement to insert the tuple `p,b` into the has-book relation, and `remove`

```
has-book(p,b)
```

as a statement to remove the tuple `p,b` from the has-book relation. Statements such as these form the basis for expressing activity in Gist specifications. They can be grouped with conventional constructs (conditional, sequential and looping statements, as the bodies of procedures, etc.). Execution of an insert or remove statement causes a *transition* from the current state to the next state, where a state comprises the set of objects and relation tuples involving those objects. Several insert and remove statements can be executed simultaneously by grouping them into an *'atomic'* statement, giving rise to a transition in which all their insertions and removals of tuples take place at once (in particular, there are no intermediary states in which some but not all of those changes have taken place). E.g.,

```
atomic { remove has-book(p1,b); insert has-book(p2,b) }
```

denotes a transition in which the tuple `p1,b` is removed from the has-book relation and the tuple `p2,b` is inserted, modeling the transfer of book `b` from person `p1 to p2`.

2.1.2 Historical queries

By default, a query takes place in the 'current' state. For example, if executing the conditional statement

```
if has-book(p,b) then ...
```

its predicate, the query of whether or not relation `has-book` holds for the tuple of person `p` and book `b`, is evaluated in the current state, that is, the state in which the conditional statement is being executed. To cause a relation query to be evaluated with respect to state s, where s is not necessarily the current state, we would issue the query `has-book(p,b) as-of s`

The general form of such a query is[2] `<predicate> as-of <state>,` denoting the evaluation of the `<predicate>` in the `<state>.` State is a pre-defined type, and can be quantified[3] over, e.g.,

```
exists (s:state) has-book(p,b) as-of s
```

asks whether the relation `has-book` holds or held on the tuple `p, b` in any state at or before the current state (but not in the future - recall that we are limiting our attention to *historical* reference.).

States can be compared, e.g.,

```
exists (s1:state,s2:state) has-book(p1,b) as-of s1 and
```

---

[2] Using angled braces to enclose nonterminals of the grammar, e.g., `<predicate>`

[3] The syntax for existential quantification is exists `(<name>: <type>, ... ) <predicate>`, where each `<name>` is an existentially quantified variable presumably used within `<predicate>`, and for universal quantification, all `(<name>: <type>, ... ) <predicate>`.

```
                          has-book(p2,b) as-of s2 and s1 < s2
```
queries whether there exist two states `s1` and `s2` such that `has-book` held of tuple `p1,b` in `s1` and of tuple `p2,b` in `s2`, and `s1` preceded `s2`, which we might paraphrase as the question: 'did `p1` have book `b` before `p2`?'

3. A GENERAL TRANSFORMATION FOR IMPLEMENTATION OF HISTORICAL REFERENCE

We first designed a simple but general transformation to implement (nearly) all historical references. This transformation is fully automatic, requiring no user intervention. In this section we describe how this transformation works, and give some simple examples.

The key to efficient implementation of historical reference is to remember sufficient information, but not overly much. Since our specification language uses relations as its predominant data structure, we designed our transformation to use whole relations as the unit of granularity when deciding what to remember.

The transformation operates in three stages - 1) determine what relations to remember; 2) introduce code to cause their values to be remembered; 3) convert historical references of those relations into conventional (non-historical) queries of the remembered information.

3.1 Determining what relations to remember

Since all historical references take the form of `<predicate> as-of <state>`, determining which relations are referenced historically is simply a matter of finding all such `as-of` references, and accumulating a list of which relations are referenced in their predicates. E.g., `has-book(p,b) as-of s` is such a reference, and its predicate references relation `has-book`.

Relations whose values are static, that is, are never changed by the specification, need no special treatment to remember their values, since their values in some historical state will be identical to their values in the current state. Analyzing the specification for relations that are never the subject of an insert or remove statement is easy, and provides a conservative approximation to identifying static relations (it is a conservative approximation in the sense that all relations that it identifies as being static are indeed static, but it is possible that there are static relations that it does not identify as such).

3.2 Remembering

Our design decision to use whole relations as the unit of granularity directs us to remember sufficient information to be able to determine all the tuples that were in any such relation in any state of the history. There are a number of choices of precisely what information to remember: at one extreme we may choose to make a copy of all a relation's tuples after every state transition - this admits the trivial retrieval of the tuples that were in that relation in any given historical state. Trading increased computation for decreased storage space, we may instead remember only the initial tuples of a relation, and subsequent changes. From this we can calculate what tuples are were in the relation in any given historical state. It is this latter style that we have implemented, and describe next.

3.2.1 Remembering incrementally

Our transformation to remember a relation works by remembering all the initial tuples of that relation, and all subsequent changes (i.e., insertion and removal of its tuples). Along with the change, we remember the state in which that change took place. To do this, we use integers to identify states, numbering from zero up. We could increment this number on every transition, but it suffices to do so on only those transitions which make a change to at least one of the historically referenced relations whose values we are remembering.[4]

The remembered information is held in several relations which the transformation adds to the specification. A unary relation, `index(integer)`, is added to hold the integer identifying the current state. For each historically referenced relation, the following relations are added:

- A relation to remember the initial tuples of that relation,
- A relation to remember insertions of tuples of that relation, and
- A relation to remember removals of tuples of that relation.

For example, if `has-book(person,book)` is a historically referenced relation, then the following relations would be added:

- `initially-has-book(person,book)` to remember the initial tuples; it holds for a person `p` and a book `b` if and only if `has-book(p,b)` held in the initial state (i.e., at the start of the computation).
- `inserted-has-book(integer,person,book)` to remember insertions of tuples; it holds for an integer `i`, a person `p` and a book `b` if and only if `has-book(p,b)` became true in the transition leading into the state represented by integer `i`.
- `removed-has-book(integer,person,book)` to remember removals of tuples; it holds for an integer `i`, a person `p` and a book `b` if and only if `has-book(p,b)` became false in the transition leading into the state represented by integer `i`.

Code is added to cause the necessary information to be remembered at the time at which it is available. Thus at the very start (in the initial state), this code must cause the copying of all the values of the historically referenced relations into their corresponding '`initially-`' relations. Changes to historically referenced relations are remembered by augmenting the statements that cause those changes with additional code to remember the change. Thus every insertion/removal of a tuple into/out of one of the historically referenced relations is augmented with code to cause that tuple to be remembered as having been inserted/removed, placing that information in the corresponding '`inserted-`'/'`removed-`' relation. Simultaneously, the integer in the `index` relation is incremented by one.

For example, if remembering the values of has-book, the following code fragment:

```
procedure check-out-book-from-library(p:person,b:book)
atomic { remove in-library(b);
         insert has-book(p,b) }
```

is transformed into:

```
procedure check-out-book-from-library(p:person,b:book)
atomic { remove in-library(b);
         if not has-book(p,b)
         then atomic { insert has-book(p,b);
                       insert inserted-has-book(index(?)+1,p,b);
                       remove index(index(?));
                       insert index(index(?)+1) } }
```

[4] Strictly speaking, we should increment this number on *every* state change, because there are a few forms of historical references that need this information, e.g., counting the number of states that have occurred between two events. All such references have what has been termed the 'stuttering' property in temporal logic (e.g., see [3]), that is, the insertion of a null transition (a transition in which no inserts or removes take place) into the behavior may change the meaning of the reference. The consensus is that references with this property are undesirable, and so we have chosen to ignore them. The advantage of taking this liberty is that there is no need to augment *every* primitive statement in the specification with code to increment the state number.

There are some subtleties in the above that need explanation:

- '`atomic { ... }`' denotes that the statements within the braces are all to be executed simultaneously (within a single transition). Thus at the same time as `has-book(p, b)` is inserted, the appropriate information is inserted into `inserted-has-book`, and `index` is changed.

- The conditional '`if not has-book(p,b) ...` ': is included because, in the semantics of our language, insert of a tuple into a relation is a no-op if the tuple is already in that relation. In such a case, for efficiency reasons we do not wish to take any action to remember information about this no-op (it would do no harm to the *correctness* of our transformation, but it would be *inefficient* - consuming unnecessary space). If we could show that the tuple is not already in the relation (e.g., perhaps the rest of the specification guarantees that the procedure `check-out-book-from-library(p,b)` is invoked only if `has-book(p,b)` does not already hold), then we could simplify the predicate of the conditional to true, and hence simplify the entire conditional to its true branch. We feel that such reasoning is best left to other transformations, and so is not incorporated as part of our historical reference removal transformation .

- `index` is the unary relation holding the integer identifying the current state. When remembering information, we must increment this integer by `1` - this is done by `remove index(index(?))`(which removes the current integer from `index`) and `insert index(index(?)+1)` (which inserts the current `integer+1` into `index`).[5]

3.2.2 Remembering incrementally and finite differencing

Remembering incremental changes to information rather than copying the information at each state is characteristic of the efficiency improvements delivered by *finite differencing* transformations [16]. We hand-coded our incremental change remembering transformation. It would be interesting to explore the alternative of using the more naive transformation that copies information at each state, and then using finite differencing to optimize the code that results.

3.3 Converting historical references

Conversion of historical references into equivalent references of the remembered relations is split into two sub-parts: 1) for each non-static historically-referenced relation R, say, introduce another relation, remembered-R, and *define* remembered-R in terms of the remembered relations introduced to remember initial values and subsequent changes; 2) re-express the historical references as equivalent, but conventional (i.e., not using historical reference) queries of these defined relations.

3.3.1 Defining the new relation in terms of the remembered information

As an example of this definition, consider the historically referenced relation `has-book(person, book)`. We add a relation `remembered-has-book(integer, person, book)`, and define it so that if `has-book(p,b)` holds in the state with integer `i`, then (and only then) `remembered-has-book(i,p,b)` will hold from then onwards. In brief, the definition works by looking for some state in which the relation held, and since which it has not been removed. This is expressed as follows:

[5] In actuality, we employ a syntactic shorthand for this kind of update.

```
remembered-has-book(s,p,b) iff
  exists(j:integer)
    ( (j=0 and initial-has-book(p,b)) or
      (0<j and j =< s and inserted-has-book(j,p,b)) ) and
      not exists (k:integer)
            (j<k and k =< s and removed-has-book(k,p,b) )
```

That is, `remember-has-book(s,p,b)` holds in the state numbered `s` if and only if there exists some state numbered `j` at or before `s` in which it held (either because it is the initial state, `j=0`, and it held initially, `initial-has-book (p, b)`, or it is a state at or before `s` whose incoming transition included an insertion of `has-book(p, b)`, i.e., `inserted-has-book(j,p,b)`) and since which it hasn't been removed, i.e., there's no state since `j`, and at or before `s`, whose incoming transition removed `has-book(p, b)`.

We have carefully crafted this as a non-recursive definition since the target language, AP5, does not support recursively defined relations (except for a few special forms). Again, there is an alternative pathway to this end, namely expressing it recursively, and relying upon recursion removal transformations.

3.3.2 Converting historical references

Historical references of non-static relations are converted into equivalent references of the corresponding defined relations.

E.g., `exists (s:state) has-book(pl.bl) as-of s`

becomes

`exists (s:integer) remembered-has-book(s,p1,bl)`

Note that the existential quantification over states has become an existential quantification over integers, our implementation's means of identifying states. In general, the type state is replaced with the type `integer`, and `R(el,e2, ... ,ei)` as-of s is replaced with `remembered-R(s,el,e2,...,ei)`.

Historical references of static relations are converted into conventional references of their current values. E.g., if title is a static relation, then

`title(bl,tl) as-of s`

becomes simply

`title(bl,tl)`.

Prior to conversion, it is necessary to normalize general forms of historical reference, `<predicate> as-of <state>`, for `<predicate>` more complex than just a relation query, into an equivalent form composed of historical references of only relation queries, `<relation-query> as-of <state>`.

For example,

`(has-book(pl,bl) and not has-book(p2.b2)) as-of s`

must be converted into

`(has-book(p1,b1) as-of s) and not (has-book(p2,b2) as-of s)`

To do this, we exhaustively apply the following set of transformation rules until all historical references have been transformed into the desired normalized form:

Conjunction

```
( <predicatel> and <predicate2> ) as-of <state>
=>
( <predicatel> as-of <state> ) and ( <predicate2> as-of
<state> )
```

Disjunction

```
( <predicatel> or <predicate2> ) as-of <state>
=>
( <predicatel> as-of <state> ) or ( <predicate2> as-of <state>
)
```

Negation

```
( not <predicate> ) as-of <state>
=>
not ( <predicate> as-of <state> )
```

Existential quantification

```
( exists ( <variable>:<type>, ... ) <predicate> ) as-of
<state>
=>
exists ( <variable>:<type>, ... ) ( <predicate> as-of <state>
)
```

Universal quantification

```
( all ( <variable>:<type>, ... ) <predicate> ) as-of <state>
=>
all ( <variable>:<type>, ... ) ( <predicate> as-of <state> )
```

Nested historical references

```
( <predicate> as-of <statel> ) as-of <state2>
=>
( <predicate> as-of <statel> ) and <statel> =< <state2>
```

The inequality in the last transformation rule arises from our restriction to historical (past, not future) references - when the inner query, `'<predicate> as-of <statel>'`, is evaluated in `<state2>`, only states at or before `<state2>` can be queried.

4. TRANSFORMATIONS FOR SPECIAL CASES

The general purpose transformation described in the previous section limits the information it remembers to just those relations that are referenced historically in the specification. This is a considerable improvement over remembering *all* historical information, however it is clear that for many special cases of historical reference, the code produced by our transformation is far from optimal - it may still be remembering unnecessarily much, and be introducing a particularly contorted computation in place of the historical references that it removes.

4.1 An example of special case historical reference removal

Consider the simple reference:

```
exists (s:state) has-book(p1,b1) as-of s
```

To answer queries of this form (for arbitrary `p1` and `b1`) we need remember only whether or not `has-book` has ever held for each `person-book` tuple. Thus we need only one additional relation, `ever-has-book(person, book)` say, initialized to hold for person `p` and book `b` if and only if `has-book(p, b)` holds in the initial state, and inserted whenever `has-book(p, b)` is inserted. E.g., the transformation of the following code fragment inserting `has-book`:

```
procedure check-out-book-from-library(p:person,b:book)
   atomic { remove in-library(b);
            insert has-book(p,b) }
```

is simply:

```
procedure check-out-book-from-library(p:person,b:book)
   atomic { remove in-library(b);
            atomic { insert has-book(p,b);
                     insert ever-has-book(p,b) } }
```

Given this, the query

```
exists (s: state) has-book(p1, b1) as-of s
```

becomes simply:

```
ever-has-book(p1,b1)
```

This is a saving in several ways - fewer relations are added, less 'remembering' code is introduced (there is no need to remember when tuples are *removed* from has-book, and there is no need to remember which state the insertions took place in), and the query that replaces the original historical reference is considerably simpler (it is no longer an existential, and does not involve a complex defined relation).

We have built a transformation that implements these '`exists (s: state) <relation-query> as-of s`' style historical references in the above manner.

4.2 The choice - special-case transformations, or follow-on optimization?

There are many other special-case forms of historical reference whose implementation can be considerably simpler than that produced by our general-purpose transformation. The question is therefore whether we should build a suite of such special-purpose transformations, or whether we should seek to optimize the code that results from the general-purpose transformation. There are pros and cons of either approach:

4.2.1 Advantages of a suite of special-case transformations

This is likely to lead to faster transformation, since the special forms can be recognized quite rapidly, and the optimal code produced directly, whereas going the other route requires producing an intermediate level of complex and somewhat verbose code which a second optimization pass must deal with. Also, it is relatively easy to build these special-case transformations, whereas the code produced by the general-purpose transformation is not particularly easy to reason about, and so we may expect the second phase of optimization to be quite tricky.

4.2.2 Advantages of optimizing the code resulting from the general-purpose transformation

The optimizations that must be performed following application of the general-purpose transformation are essentially simplifications of data structures (and the code that creates/modifies them) that take advantage of the limited ways in which those structures are updated and accessed. Such optimization capabilities would be valuable for all data structures, not only those resulting from a historical-reference-removing transformation. Indeed, there is some justification to the argument that to put together a reasonably complete transformation system, we would have to build this capability anyway, in which case constructing a suite of special-case transformations would be a duplication of effort.

4.3 The optimum, the reality

We believe that the optimum solution would be to have a separate data-structure optimizing transformation able to take the output of our general purpose historical removal transformation, and produce code of comparable efficiency to the hand-crafted special-purpose transformations. Furthermore, if the speed of the general purpose historical removal transformation followed by this data-structure optimizing transformation (i.e., the speed at which they transform, not the speed of the transformed code they produce) is inferior to historical removal via special-purpose transformations, then those special-purpose transformations should be derived from the combination of these two transformations partially applied to the special cases of historical reference. It would be interesting to try to gauge the feasibility of this approach given the current capabilities of partial evaluation techniques. The partial evaluation paradigm has been applied to the complex problem of producing parsers from parser generators by fixing the grammar (e.g., [11]), so perhaps it would be viable for this application.

In practice, we do *not* have a powerful data-structure optimizing transformation, and so have followed the route of building up a suite of special-purpose historical removal transformations to deal with those recurring idioms of historical reference whose implementation via our general historical removal transformation we deem unacceptably inefficient.

5. TARGETING TO THE CAPABILITIES OF THE TARGET LANGUAGE

Each of our historical removal transformations introduces code to cause information to be remembered as it arises. The transformations work by traversing the statements of the specification, looking for those that insert/remove tuples into/out of historically referenced relations, and augmenting those statements with further statements which will run simultaneously and remember the appropriate information. This is essentially a 'compilation' of the necessary remembering activity into the specification. An alternative is to rely upon a more interpretive mechanism that supervises each state change.

Such a mechanism exists in our target language, AP5 [8]. This mechanism is an interpretive cycle that, on every state change, checks the consistency of the prospective new state with respect to a set of 'consistency rules'. On detection of a state change that would lead to inconsistency, 'repair rules' are applied to (try to) alter the state change so as to lead to a consistent state. Briefly, an AP5 repair rule works by proposing further activity (inserts and/or removes of tuples) to be done simultaneously with the activity of the transition that is causing the consistency rule violation. We can use this to define the code that causes information (relation tuples) to be remembered, by defining as a consistency rule the condition that some event has occurred that requires information to be remembered, and yet that information has not yet been remembered; the 'repair' of this is to cause the appropriate information to be remembered.

5.1 An example of consistency and repair rules to cause remembering

Consider the special-case transformation for removing historical references of the form `exists (s:state) has-book(p,b) as-of s.`

This worked by transforming every insert of a tuple into the `has-book` relation so as to simultaneously insert the `ever-has-book` relation on the same tuple, e.g.,

```
insert has-book(p,b)
```

was transformed to

```
atomic { insert has-book(p,b); insert ever-has-book(p,b) }
```

Using the AP5 consistency & repair rule mechanism, we define the consistency rule: `exists (p:person,b:book) has-book(p,b) and not ever-has-book(p,b)`

(i.e., `has-book` holds of the tuple `p,b,` but `ever-has-book` does not hold) and the corresponding repair rule:

```
insert ever-has-book(p,b)
```

(i.e., insert `ever-has-book` to hold of that tuple). Thus whenever `has-book` is inserted of some tuple, the effect of this consistency & repair is to ensure that `ever-has-book` is also inserted if it doesn't already hold.

In general, our historical reference removal transformations all work by remembering information in new relations introduced specially to hold such information. Hence it is always possible to express the check for whether information needs to be remembered as a consistency rule that examines the changes taking place, and the information already remembered. Likewise, it is always possible to express the code to cause information to be remembered as a repair rule in response to violations of these consistency rules.

5.2 Using the AP5 consistency cycle vs. compilation

The advantages of using the AP5 consistency cycle mechanism in this manner stem from the remembering code being expressed simply as a consistency rule together with an associated repair. Thus the transformations can operate more speedily (they need not scan the entire specification for statements that might require augmentation), and the resulting code is more compact (extra statements to remember information have not been scattered through the specification).

Conversely, the resulting code may be less efficient, since the burden is placed on the consistency cycle, an operation run on every state change. As a mitigating factor, we would point out that considerable effort has been invested in the AP5 implementation to make this cycle efficient.

5.3 The route from Gist to AP5's consistency cycle

We have described the targeting toward AP5's consistency cycle and repair mechanism as if we were transforming Gist specifications directly into AP5. In truth, our historical reference removal transformations operate by transforming Gist specifications using historical reference into Gist specifications not using historical reference, and it is the latter which are later transformed into AP5. It is useful to be able to stay within Gist so that we may apply other Gist-to-Gist transformations even after historical references have been removed - this offers more flexibility in sequencing transformations.

Unfortunately, Gist has the notion of consistency, but no obvious counterpart of AP5's repair rules; a Gist specification denotes those and only those behaviors whose states are all consistent (with respect to a set of consistency conditions we call 'constraints'), but apparently has no notion of reacting to (and repairing) a transition that is leading to an inconsistent state. Gist does, however, have 'demons' - activity invoked in response to the state meeting some condition, and has *future* reference. Thus we can write the remembering code as a demon which looks at the *next* state to see if the condition that necessitates remembering information will occur, and looks at the current state to see if the information has not already been remembered - if so, it invokes the activity to remember the information, this activity to be done immediately (i.e., simultaneously with whatever change is going on).

Thus the path from Gist with historical reference to AP5 (using AP5's consistency cycle to cause information to be remembered) goes through an intermediate level of Gist with no historical reference, but somewhat surprisingly, with some (albeit very limited forms of) future reference! This is an example of *partial* overlap between the semantic constructs of two languages - we do not know how to convert *arbitrary* uses of future reference in Gist into AP5, yet the very special forms of future reference that we introduce through our historical reference removing transformations can all be translated into AP5 (specifically, into uses of the consistency & repair rule mechanism).

6. EXTENDING THE SPECIFICATION LANGUAGE EXPRESSIONS OF HISTORICAL REFERENCES

In what we have described so far, the only way of expressing a historical reference has been to use the `<predicate> as-of <state>` syntax. For ease of specification, we have found it useful to provide some more convenient notations for expressing historical references. We use automatic translation to cause these to be converted into the basic 'kernel' form of historical reference, which our historical reference removing transformations know how to deal with. We give some simple examples, and follow with a discussion of the relevance of this to transformation.

We define `whenever <predicate>` to denote the states in which `<predicate>` is true. E.g.,

```
has-book(p,b) as-of whenever true
```

is translated into:

```
exists (s:state) has-book(p,b) as-of s.
```

In a similar manner, we define 'intervals' with the syntax

```
during (<state>, <state>)
```

e.g.,

```
has-book(p,b) as-of during (s1, s2)
```

is translated into

```
exists (s:state) (has-book(p,b) as-of s) and (s1 < s) and (s < s2)
```

Like mathematical intervals, their ends may be be 'open', i.e., not including their endpoints (states) (as above) or 'closed', e.g.,

```
has-book(p,b) as-of during [s1,s2]
```

which translates into:

```
exists (s:state)
   (has-book(p,b) as-of s) and (s1 =< s) and (s =< s2)
```

(note the '=<' in place of '<').

What is of interest is not the particular forms of expression that we have provided (indeed, we have had insufficient experience with their use to say whether they are worthy of retention), but rather the way in which we make them available. The key to our approach is to build another language layer on top of the 'kernel' form of historical reference expression (the `<predicate> as-of <state>` form), and to use translators (a stylized form of automatic transformation - see [22] for details) to convert the new notations into uses of the 'kernel' form. (We remark that translators are also used to convert a subset of Gist into AP5.) The advantage of this approach is a convenient and easily modifiable syntax for expressing historical reference, where the only adjustments required are to the translators that automatically convert such syntax into the established 'kernel' form. Thus the transformations for removal of historical reference can continue to work off of the 'kernel' form, and need not be adapted to deal with the new syntax.

As in the issue of general-purpose vs. special-purpose historical reference removal transformations, we can see a tradeoff here too. Translation into the 'kernel' form represents a bottleneck through which all removal of historical references must pass. In particular, special cases of historical reference that are expressed concisely using these definitional extensions may translate into rather verbose 'kernel' layer equivalents. The alternative would be to redefine the special-purpose historical removal transformations to operate on the expressive language constructs that we have added. For this choice we perceive that the disadvantages of going through the intermediate layer (slower operation of transformations, and perhaps a more verbose expression of those transformations) are far outweighed by the advantages (the 'kernel' layer of historical reference expression is expected to remain relatively stable, whereas the extensions that we provide by translation may well change and extend over time, so by having the historical removal transformations operate upon the 'kernel' layer, we decouple them from these more frequent changes).

7. RELATED WORK

Historical reference in particular has been used in the contexts of:

- programming languages: e.g., Lucid provides the ability to refer to the sequence of values of a variable [2],

- databases: generally the assumption has been that *all* past states of the data are retained in one form or another in order to be able to answer any query that the user might issue. The emphasis of much of this research has been to establish the semantics, and study how to query such a database. For example, a simple temporal data model is presented in [1], and a user-friendly historical query language is shown in [20].

- programming environments: e.g., the INTERLISP environment recorded a history of interaction with the user to permit the undoing and redoing of past commands [21].

The historical reference implementation work most closely related work to ours is:

- In the context of term-rewriting systems, constraints expressed using historical reference to the evaluation sequence have been proposed as a means to limit the possible rewriting behaviors [9]. As in our approach, historical references are automatically transformed into non-historical references by remembering extra information. Whereas their focus is to constrain evaluations, ours is a more general goal of supporting arbitrary queries of past states' information.

- A specification language with historical reference to the sequence of 'events' (communications between processes) and a prototyping environment which remembers sufficient information to answer historical references, are described in [10]. Their temporal language appears designed to facilitate transformation. Similar research is reported in [12], although here the transformation towards an efficient non-historical implementation is done by first tailoring the historical references, and thereafter translating to a non-historical form; the first phase appears to require the hand-directed application of transformations.

Transformation work is very extensive, as indicated by the contents of [15] and of this volume. The following are of particular relevance:

- Language extension by definitional transformations [13],

- Kestrel and Reasoning Systems Inc.'s compiler of a high-level language in a transformational style [18, 17],

- Boyle's transformations between language levels, and the role of normal forms in this activity [7, 6],

- Wile's translations between sub-languages [22], and

- the definition of the wide-spectrum language CIP-L by transformation to a simple kernel. [5]

The common feature of all of these efforts is the decomposition of the transformation problem based on the language layers (or pieces), a phenomenon that clearly arose in our historical reference removal transformation.

8. CONCLUSIONS

We have argued that historical reference is a specification freedom that can be successfully implemented through transformation. Our general purpose transformation does this automatically.

While we took some steps to ensure that the code produced by this transformation is reasonably efficient (e.g., remembering initial values of only those relations that are referenced historically, and subsequent changes to those relations), we recognized that many special cases of historical reference admit to more efficient implementations. To deal with this we have embarked on the course of building a suite of special-purpose historical reference removal transformations (also automatic), although we recognize the desirability of the alternative of building a powerful data-structure optimizer to be run as a follow-on transformation to the general-purpose transformation. This choice – of the level at which to build transformations – recurs frequently in program transformation.

Another choice that was open to us was whether or not to use AP5's consistency cycle mechanism, which offers an interpretive-like implementation option for part of the historical reference implementations. As very high level programming languages become populated with such sophisticated mechanisms, targeting transformations to these languages will offer more choices of this nature.

We note that have made repeated use of 'layers' of transformation, from Gist specifications using extended notations for historical reference down to the 'kernel' forms of historical reference, in turn to Gist with no historical reference, and finally to AP5 (our target language which can be executed). Such layering is a commonly used technique for structuring transformational developments.

9. ACKNOWLEDGEMENTS

Implementation of these transformations has made use of David Wile's POPART system and our group's AP5 system, primarily the work of Don Cohen and Neil Goldman. Lewis Johnson has merged the two to produce an eminently usable transformation environment. Our entire group, headed by Bob Balzer, has provided the rich context in which this work has taken place.

10. REFERENCES

[1] G. Ariav. A temporally oriented data model. *ACM Transactions on Database' Systems,* 11(4):499-527, December 1986.

[2] E.A. Ashcroft and W.W. Wadge. Lucid, a nonprocedural language with iteration. *CACM,* 20(7):519-526, July 1977.

[3] R.J.R Back and R Kurki-Suonio. Distributed cooperation with action systems. *ACM TOPLAS,* 10(4):513-554, 1988.

[4] R. Balzer, D. Cohen, M.S. Feather, N.M. Goldman, W. Swartout, and D.S. Wile. Operational specification as the basis for specification validation. In *Theory and practice of software technology,* pages 21-49. North-Holland, 1983.

[5] F.L. Bauer et al. *The Munich project CJP. Volume J: The wide spectrum language CIP-L.,* volume 183 of *Lecture Notes in Computer Science.* Springer, 1985.

[6] J.M. Boyle, K.W. Dritz, M.N. Muralidharan, and R.J. Taylor. Deriving sequential and parallel programs from pure lisp specifications by program transformation. In L.G.L.T. Meertens, editor, *Proceedings of the IFIP TC2 Working Conference on Program Specification and Transformation, Bad Toelz, FRG,* 1986, pages 1-19. North-Holland, 1987.

[7] J.M. Boyle and M.N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering,* SE-10(5):574-588, 1984.

[8] D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data, Portland, Oregon,* pages 225-234.

ACM Press, 1989. SIGMOD RECORD Volume 18, Number 2, June 1989.

[9] J. Darlington and L. While. Controlling the behavior of functional language systems. In *Proceedings, Functional Programming Languages and Computer Architectures Conference, Portland, Oregon, USA,* pages 278-300, 1987.

[10] N. de Francesco and G. Vaglini. Description of a tool for specifying and prototyping concurrent programs. *IEEE Transactions on Software Engineering, 14(11):1554*1564, November 1988.

[11] A.P. Ershov and B.N. Ostrovski. Controlled mixed computation and its application to systematic development of language oriented parsers. In L.G.L.T. Meertens, editor, *Proceedings of the IFI? TC2 Working Conference on Program Specification and Transformation, Bad Toelz, FRG,* 1986, pages 31-48. North-Holland, 1987.

[12] J. Jaray. Timed specifications for the development of real-time systems. In *Proceedings, Symposium on Formal Techniques in Real- Time and Fault- Tolerant Systems, Warwick, UK, September* 1988, Lecture notes in computer science, No. 331, pages 67-83. Springer-Verlag, 1988.

[13] T.E. Cheatham Jr. Reusability through program transformations. *IEEE Transactions on Software Engineering,* SE-I0(5):589-594, September 1984.

[14] P.E. London and M.S. Feather. Implementing specification freedoms. In C. Rich and R. Waters., editors, *Readings in Artificial Intelligence and Software Engineering,* pages 285-305. Morgan Kaufmann, 1986. Originally published in Science of Computer Programming, 1982 No.2, pp 91-131.

[15] L.G.L.T. Meertens, editor. *Proceedings of the IFIP TC2 Working Conference on Program Specification and Transformation, Bad Toelz, FRG,* 1986. North-Holland, 1987.

[16] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems,* 4(3):402-454, July 1982.

[17] Reasoning Systems Inc. *Refine User's Guide, 1989.*

[18] D.R. Smith, G.B. Kotik, and S.J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering,* SE-11(11):1278-1295, 1985.

[19] G.L. Steele. *Common Lisp: the language.* Digital Press, 1984.

[20] A.D. Tansel, M.E. Arkun, and G. Ozsoyoglu. Time-by-example query language for historical databases. *IEEE Transactions on Software Engineering, 15(4):464-478,* April 1989.

[21] W. Teitleman. *Interlisp reference manual.* Xerox Palo Alto Research Center, 1978.

[22] D.S. Wile. Local formalisms: Widening the spectrum of wide-spectrum languages.
In L.G.L.T. Meertens, editor, *Proceedings of the IPIP TC2 Working Conference on Program Specification and Transformation, Bad Toelz, PRG,* 1986, pages 459-481. North-Holland, 1987.